13 | API 风格（下）：RPC API介绍

2021-06-24 孔令飞

《Go 语言项目开发实战》 课程介绍 >

讲述：孔令飞
时长 10:53 大小 9.97M

你好，我是孔令飞。这一讲，我们继续来看下如何设计应用的 API 风格。

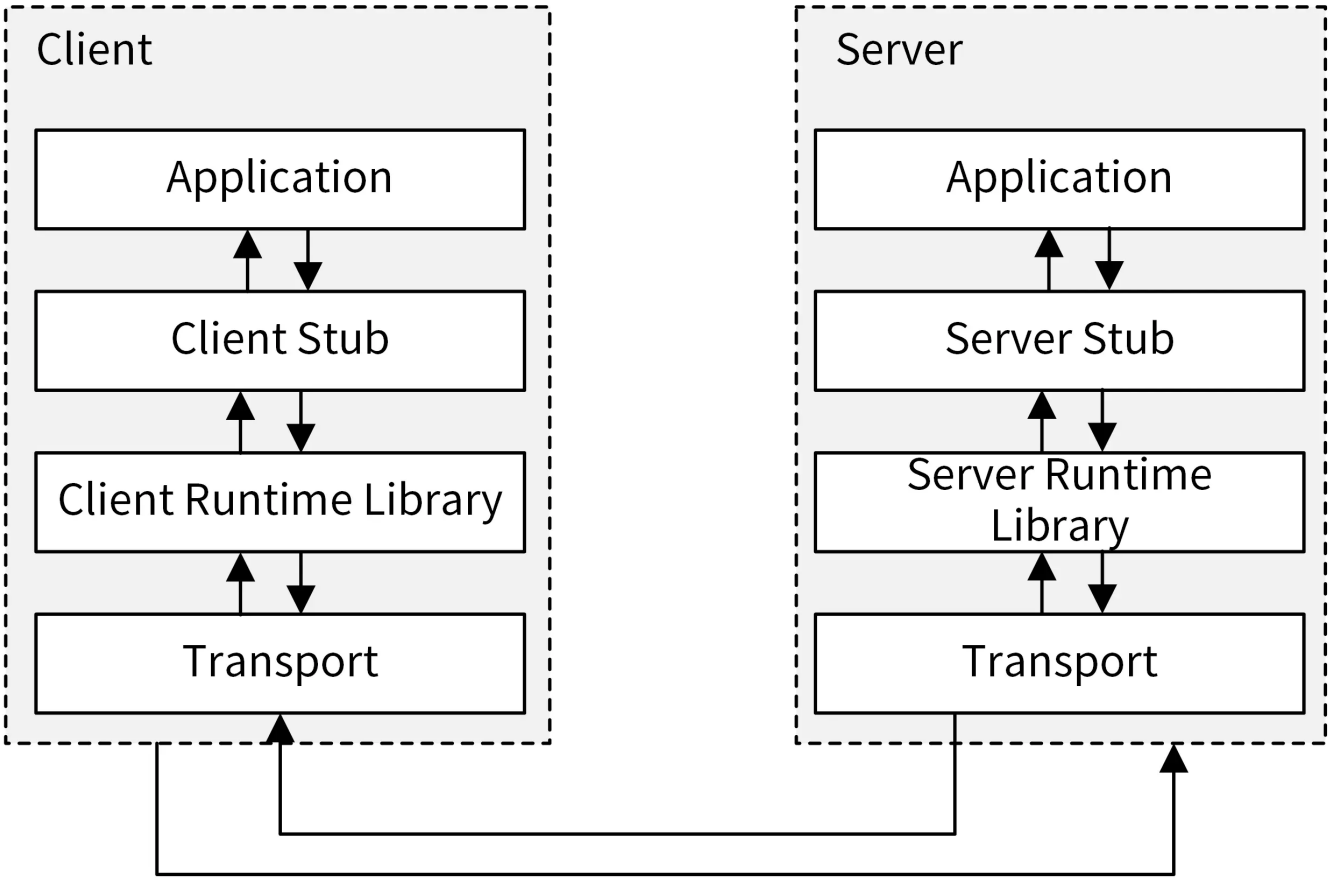
上一讲，我介绍了 REST API 风格，这一讲我来介绍下另外一种常用的 API 风格，RPC。在 Go 项目开发中，如果业务对性能要求比较高，并且需要提供给多种编程语言调用，这时候就可以考虑使用 RPC API 接口。RPC 在 Go 项目开发中用得也非常多，需要我们认真掌握。

RPC 介绍

根据维基百科的定义，RPC（Remote Procedure Call），即远程过程调用，是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员不用额外地为这个交互作用编程。

通俗来讲，就是服务端实现了一个函数，客户端使用 RPC 框架提供的接口，像调用本地函数一样调用这个函数，并获取返回值。RPC 屏蔽了底层的网络通信细节，使得开发人员无需关注网络编程的细节，可以将更多的时间和精力放在业务逻辑本身的实现上，从而提高开发效率。

RPC 的调用过程如下图所示：



RPC 调用具体流程如下：

1. Client 通过本地调用，调用 Client Stub。
2. Client Stub 将参数打包（也叫 Marshalling）成一个消息，然后发送这个消息。
3. Client 所在的 OS 将消息发送给 Server。
4. Server 端接收到消息后，将消息传递给 Server Stub。
5. Server Stub 将消息解包（也叫 Unmarshalling）得到参数。
6. Server Stub 调用服务端的子程序（函数），处理完后，将最终结果按照相反的步骤返回给 Client。

这里需要注意，Stub 负责调用参数和返回值的流化（serialization）、参数的打包和解包，以及网络层的通信。Client 端一般叫 Stub，Server 端一般叫 Skeleton。

目前，业界有很多优秀的 RPC 协议，例如腾讯的 Tars、阿里的 Dubbo、微博的 Motan、Facebook 的 Thrift、RPCX，等等。但使用最多的还是 gRPC，这也是本专栏所采用的 RPC 框架，所以接下来我会重点介绍 gRPC 框架。

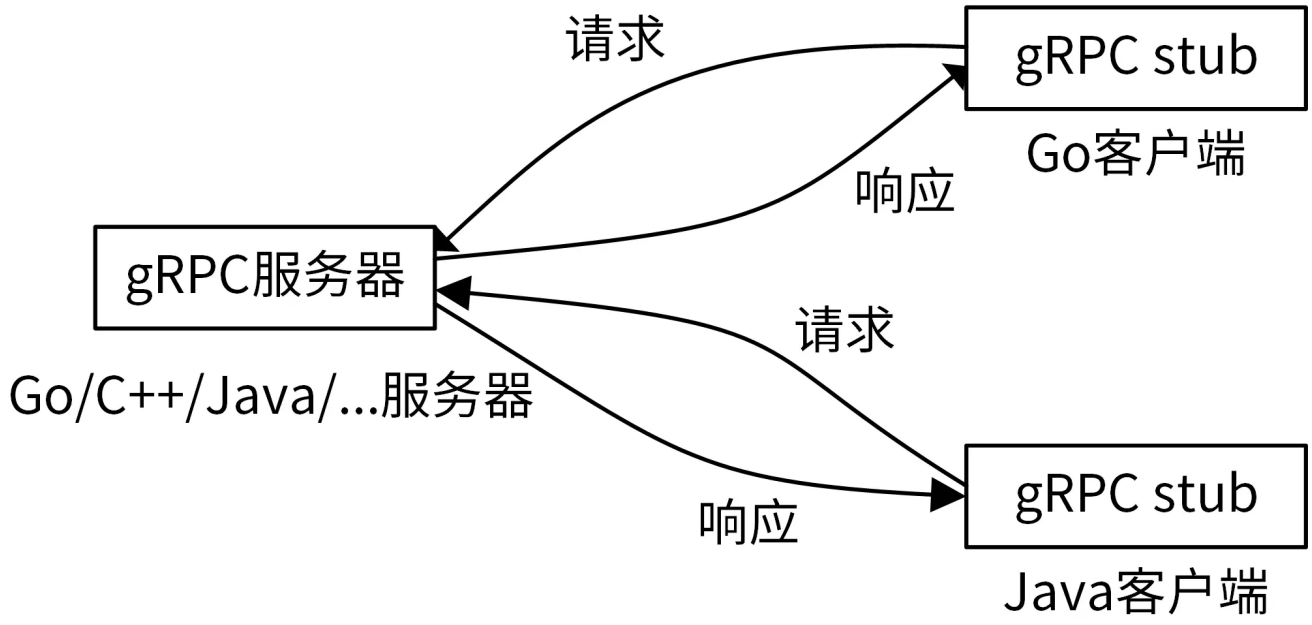
gRPC 介绍

gRPC 是由 Google 开发的高性能、开源、跨多种编程语言的通用 RPC 框架，基于 HTTP 2.0 协议开发，默认采用 Protocol Buffers 数据序列化协议。gRPC 具有如下特性：

- 支持多种语言，例如 Go、Java、C、C++、C#、Node.js、PHP、Python、Ruby 等。
- 基于 IDL（Interface Definition Language）文件定义服务，通过 proto3 工具生成指定语言的数据结构、服务端接口以及客户端 Stub。通过这种方式，也可以将服务端和客户端解耦，使客户端和服务端可以并行开发。
- 通信协议基于标准的 HTTP/2 设计，支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等特性。
- 支持 Protobuf 和 JSON 序列化数据格式。Protobuf 是一种语言无关的高性能序列化框架，可以减少网络传输流量，提高通信效率。

这里要注意的是，gRPC 的全称不是 golang Remote Procedure Call，而是 google Remote Procedure Call。

gRPC 的调用如下图所示：



在 gRPC 中，客户端可以直接调用部署在不同机器上的 gRPC 服务所提供的方法，调用远端的 gRPC 方法就像调用本地的方法一样，非常简单方便，通过 gRPC 调用**，我们可以非常容易地构建出一个分布式应用。**

像很多其他的 RPC 服务一样，gRPC 也是通过 IDL 语言，预先定义好接口（接口的名字、传入参数和返回参数等）。在服务端，gRPC 服务实现我们所定义的接口。在客户端，gRPC 存根提供了跟服务端相同的方法。

gRPC 支持多种语言，比如我们可以用 Go 语言实现 gRPC 服务，并通过 Java 语言客户端调用 gRPC 服务所提供的方法。通过多语言支持，我们编写的 gRPC 服务能满足客户端多语言的需求。

gRPC API 接口通常使用的数据传输格式是 Protocol Buffers。接下来，我们就一起了解下 Protocol Buffers。

Protocol Buffers 介绍

Protocol Buffers（ProtocolBuffer/ protobuf）是 Google 开发的一套对数据结构进行序列化的方法，可用作（数据）通信协议、数据存储格式等，也是一种更加灵活、高效的数据格式，与 XML、JSON 类似。它的传输性能非常好，所以常被用在一些对数据传输性能要求比较高的系统中，作为数据传输格式。Protocol Buffers 的主要特性有下面这几个。

更快的数据传输速度：protobuf 在传输时，会将数据序列化为二进制数据，和 XML、JSON 的文本传输格式相比，这可以节省大量的 IO 操作，从而提高数据传输速度。

跨平台多语言：protobuf 自带的编译工具 protoc 可以基于 protobuf 定义文件，编译出不同语言的客户端或者服务端，供程序直接调用，因此可以满足多语言需求的场景。

具有非常好的扩展性和兼容性，可以更新已有的数据结构，而不破坏和影响原有的程序。

基于 IDL 文件定义服务，通过 proto3 工具生成指定语言的数据结构、服务端和客户端接口。

在 gRPC 的框架中，Protocol Buffers 主要有三个作用。

第一，可以用来定义数据结构。举个例子，下面的代码定义了一个 SecretInfo 数据结构：

复制代码

```
// SecretInfo contains secret details.
message SecretInfo {
    string name = 1;
    string secret_id  = 2;
    string username   = 3;
    string secret_key = 4;
    int64 expires = 5;
    string description = 6;
    string created_at = 7;
    string updated_at = 8;
}
```

第二，可以用来定义服务接口。下面的代码定义了一个 Cache 服务，服务包含了 ListSecrets 和 ListPolicies 两个 API 接口。

复制代码

```
// Cache implements a cache rpc service.
service Cache{
  rpc ListSecrets(ListSecretsRequest) returns (ListSecretsResponse) {}
  rpc ListPolicies(ListPoliciesRequest) returns (ListPoliciesResponse) {}
}
```

第三，可以通过 protobuf 序列化和反序列化，提升传输效率。

gRPC 示例

我们已经对 gRPC 这一通用 RPC 框架有了一定的了解，但是你可能还不清楚怎么使用 gRPC 编写 API 接口。接下来，我就通过 gRPC 官方的一个示例来快速给大家展示下。运行本示例需要在 Linux 服务器上安装 Go 编译器、Protocol buffer 编译器（protoc，v3）和 protoc 的 Go 语言插件，在 **02 讲** 中我们已经安装过，这里不再讲具体的安装方法。

这个示例分为下面几个步骤：

1. 定义 gRPC 服务。
2. 生成客户端和服务器代码。
3. 实现 gRPC 服务。
4. 实现 gRPC 客户端。

示例代码存放在gopractise-demo/apistyle/greeter目录下。代码结构如下：

复制代码

```
$ tree
├── client
│   └── main.go
├── helloworld
│   ├── helloworld.pb.go
│   └── helloworld.proto
└── server
    └── main.go
```

client 目录存放 Client 端的代码，helloworld 目录用来存放服务的 IDL 定义，server 目录用来存放 Server 端的代码。

下面我具体介绍下这个示例的四个步骤。

1. 定义 gRPC 服务。

首先，需要定义我们的服务。进入 helloworld 目录，新建文件 helloworld.proto：

复制代码

```
$ cd helloworld
$ vi helloworld.proto
```

内容如下：

复制代码

```
syntax = "proto3";

option go_package = "github.com/marmotedu/gopractise-demo/apistyle/greeter/hel

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

在 helloworld.proto 定义文件中，option 关键字用来对.proto 文件进行一些设置，其中 go_package 是必需的设置，而且 go_package 的值必须是包导入的路径。package 关键字指定生成的.pb.go 文件所在的包名。我们通过 service 关键字定义服务，然后再指定该服务拥有的 RPC 方法，并定义方法的请求和返回的结构体类型：

复制代码

```
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

gRPC 支持定义 4 种类型的服务方法，分别是简单模式、服务端数据流模式、客户端数据流模式和双向数据流模式。

> 简单模式（Simple RPC）：是最简单的 gRPC 模式。客户端发起一次请求，服务端响应一个数据。定义格式为 rpc SayHello (HelloRequest) returns (HelloReply) {}。
>
> 服务端数据流模式（Server-side streaming RPC）：客户端发送一个请求，服务器返回数据流响应，客户端从流中读取数据直到为空。定义格式为 rpc SayHello (HelloRequest) returns (stream HelloReply) {}。
>
> 客户端数据流模式（Client-side streaming RPC）：客户端将消息以流的方式发送给服务器，服务器全部处理完成之后返回一次响应。定义格式为 rpc SayHello (stream HelloRequest) returns (HelloReply) {}。
>
> 双向数据流模式（Bidirectional streaming RPC）：客户端和服务端都可以向对方发送数据流，这个时候双方的数据可以同时互相发送，也就是可以实现实时交互 RPC 框架原理。定义格式为 rpc SayHello (stream HelloRequest) returns (stream HelloReply) {}。

本示例使用了简单模式。.proto 文件也包含了 Protocol Buffers 消息的定义，包括请求消息和返回消息。例如请求消息：

复制代码

```
// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}
```

2. 生成客户端和服务器代码。

接下来，我们需要根据.proto 服务定义生成 gRPC 客户端和服务器接口。我们可以使用 protoc 编译工具，并指定使用其 Go 语言插件来生成：

复制代码

```
$ protoc -I. --go_out=plugins=grpc:$GOPATH/src helloworld.proto
$ ls
helloworld.pb.go  helloworld.proto
```

你可以看到，新增了一个 helloworld.pb.go 文件。

3. 实现 gRPC 服务。

接着，我们就可以实现 gRPC 服务了。进入 server 目录，新建 main.go 文件：

复制代码

```
$ cd ../server
$ vi main.go
```

main.go 内容如下：

复制代码

```
// Package main implements a server for Greeter service.
package main

import (
  "context"
  "log"
  "net"

  pb "github.com/marmotedu/gopractise-demo/apistyle/greeter/helloworld"
  "google.golang.org/grpc"
)

const (
  port = ":50051"
)

// server is used to implement helloworld.GreeterServer.
type server struct {
  pb.UnimplementedGreeterServer
}

// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.Hello
  log.Printf("Received: %v", in.GetName())
  return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil
}

func main() {
  lis, err := net.Listen("tcp", port)
  if err != nil {
    log.Fatalf("failed to listen: %v", err)
  }
```

```
33   s := grpc.NewServer()
34   pb.RegisterGreeterServer(s, &server{})
35   if err := s.Serve(lis); err != nil {
36     log.Fatalf("failed to serve: %v", err)
37   }
38 }
```

上面的代码实现了我们上一步根据服务定义生成的 Go 接口。

我们先定义了一个 Go 结构体 server，并为 server 结构体添加 `SayHello(context.Context, pb.HelloRequest) (pb.HelloReply, error)` 方法，也就是说 server 是 GreeterServer 接口（位于 helloworld.pb.go 文件中）的一个实现。

在我们实现了 gRPC 服务所定义的方法之后，就可以通过 `net.Listen(...)` 指定监听客户端请求的端口；接着，通过 `grpc.NewServer()` 创建一个 gRPC Server 实例，并通过 `pb.RegisterGreeterServer(s, &server{})` 将该服务注册到 gRPC 框架中；最后，通过 `s.Serve(lis)` 启动 gRPC 服务。

创建完 main.go 文件后，在当前目录下执行 `go run main.go` ，启动 gRPC 服务。

4. 实现 gRPC 客户端。

打开一个新的 Linux 终端，进入 client 目录，新建 main.go 文件：

复制代码

```
$ cd ../client
$ vi main.go
```

main.go 内容如下：

复制代码

```
// Package main implements a client for Greeter service.
package main

import (
  "context"
```

```
  "log"
  "os"
  "time"

  pb "github.com/marmotedu/gopractise-demo/apistyle/greeter/helloworld"
  "google.golang.org/grpc"
)

const (
  address     = "localhost:50051"
  defaultName = "world"
)

func main() {
  // Set up a connection to the server.
  conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
  if err != nil {
    log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close()
  c := pb.NewGreeterClient(conn)

  // Contact the server and print out its response.
  name := defaultName
  if len(os.Args) > 1 {
    name = os.Args[1]
  }
  ctx, cancel := context.WithTimeout(context.Background(), time.Second)
  defer cancel()
  r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})
  if err != nil {
    log.Fatalf("could not greet: %v", err)
  }
  log.Printf("Greeting: %s", r.Message)
}
```

在上面的代码中，我们通过如下代码创建了一个 gRPC 连接，用来跟服务端进行通信：

复制代码

```
// Set up a connection to the server.
conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
if err != nil {
    log.Fatalf("did not connect: %v", err)
}
defer conn.Close()
```

在创建连接时，我们可以指定不同的选项，用来控制创建连接的方式，例如 grpc.WithInsecure()、grpc.WithBlock() 等。gRPC 支持很多选项，更多的选项可以参考 grpc 仓库下 dialoptions.go 文件中以 With 开头的函数。

连接建立起来之后，我们需要创建一个客户端 stub，用来执行 RPC 请求`c := pb.NewGreeterClient(conn)`。创建完成之后，我们就可以像调用本地函数一样，调用远程的方法了。例如，下面一段代码通过 `c.SayHello` 这种本地式调用方式调用了远端的 SayHello 接口：

复制代码

```
r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})
if err != nil {
    log.Fatalf("could not greet: %v", err)
}
log.Printf("Greeting: %s", r.Message)
```

从上面的调用格式中，我们可以看到 RPC 调用具有下面两个特点。

> 调用方便：RPC 屏蔽了底层的网络通信细节，使得调用 RPC 就像调用本地方法一样方便，调用方式跟大家所熟知的调用类的方法一致：`ClassName.ClassFuc(params)`。
>
> 不需要打包和解包：RPC 调用的入参和返回的结果都是 Go 的结构体，不需要对传入参数进行打包操作，也不需要对返回参数进行解包操作，简化了调用步骤。

最后，创建完 main.go 文件后，在当前目录下，执行 go run main.go 发起 RPC 调用：

复制代码

```
$ go run main.go
2020/10/17 07:55:00 Greeting: Hello world
```

至此，我们用四个步骤，创建并调用了一个 gRPC 服务。接下来我再给大家讲解一个在具体场景中的注意事项。

在做服务开发时，我们经常会遇到一种场景：定义一个接口，接口会通过判断是否传入某个参数，决定接口行为。例如，我们想提供一个 GetUser 接口，期望 GetUser 接口在传

入 username 参数时，根据 username 查询用户的信息，如果没有传入 username，则默认根据 userId 查询用户信息。

这时候，我们需要判断客户端有没有传入 username 参数。我们不能根据 username 是否为空值来判断，因为我们不能区分客户端传的是空值，还是没有传 username 参数。这是由 Go 语言的语法特性决定的：如果客户端没有传入 username 参数，Go 会默认赋值为所在类型的零值，而字符串类型的零值就是空字符串。

那我们怎么判断客户端有没有传入 username 参数呢？最好的方法是通过指针来判断，如果是 nil 指针就说明没有传入，非 nil 指针就说明传入，具体实现步骤如下：

1. 编写 protobuf 定义文件。

新建 user.proto 文件，内容如下:

复制代码

```
syntax = "proto3";

package proto;
option go_package = "github.com/marmotedu/gopractise-demo/protobuf/user";

//go:generate protoc -I. --experimental_allow_proto3_optional --go_out=plugins

service User {
  rpc GetUser(GetUserRequest) returns (GetUserResponse) {}
}

message GetUserRequest {
  string class = 1;
  optional string username = 2;
  optional string user_id = 3;
}

message GetUserResponse {
  string class = 1;
  string user_id = 2;
  string username = 3;
  string address = 4;
  string sex = 5;
  string phone = 6;
}
```

你需要注意，这里我们在需要设置为可选字段的前面添加了 **optional** 标识。

2. 使用 protoc 工具编译 protobuf 文件。

在执行 protoc 命令时，需要传入`--experimental_allow_proto3_optional`参数以打开 **optional** 选项，编译命令如下：

复制代码

```
$ protoc --experimental_allow_proto3_optional --go_out=plugins=grpc:. user.pro
```

上述编译命令会生成 user.pb.go 文件，其中的 GetUserRequest 结构体定义如下：

复制代码

```
type GetUserRequest struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Class    string  `protobuf:"bytes,1,opt,name=class,proto3" json:"class,omi
    Username *string `protobuf:"bytes,2,opt,name=username,proto3,oneof" json:"
    UserId   *string `protobuf:"bytes,3,opt,name=user_id,json=userId,proto3,on
}
```

通过 `optional` + `--experimental_allow_proto3_optional` 组合，我们可以将一个字段编译为指针类型。

3. 编写 gRPC 接口实现。

新建一个 user.go 文件，内容如下：

复制代码

```
package user

import (
    "context"

    pb "github.com/marmotedu/api/proto/apiserver/v1"

```

```
    "github.com/marmotedu/iam/internal/apiserver/store"
)

type User struct {
}

func (c *User) GetUser(ctx context.Context, r *pb.GetUserRequest) (*pb.GetUser
    if r.Username != nil {
        return store.Client().Users().GetUserByName(r.Class, r.Username)
    }

    return store.Client().Users().GetUserByID(r.Class, r.UserId)
}
```

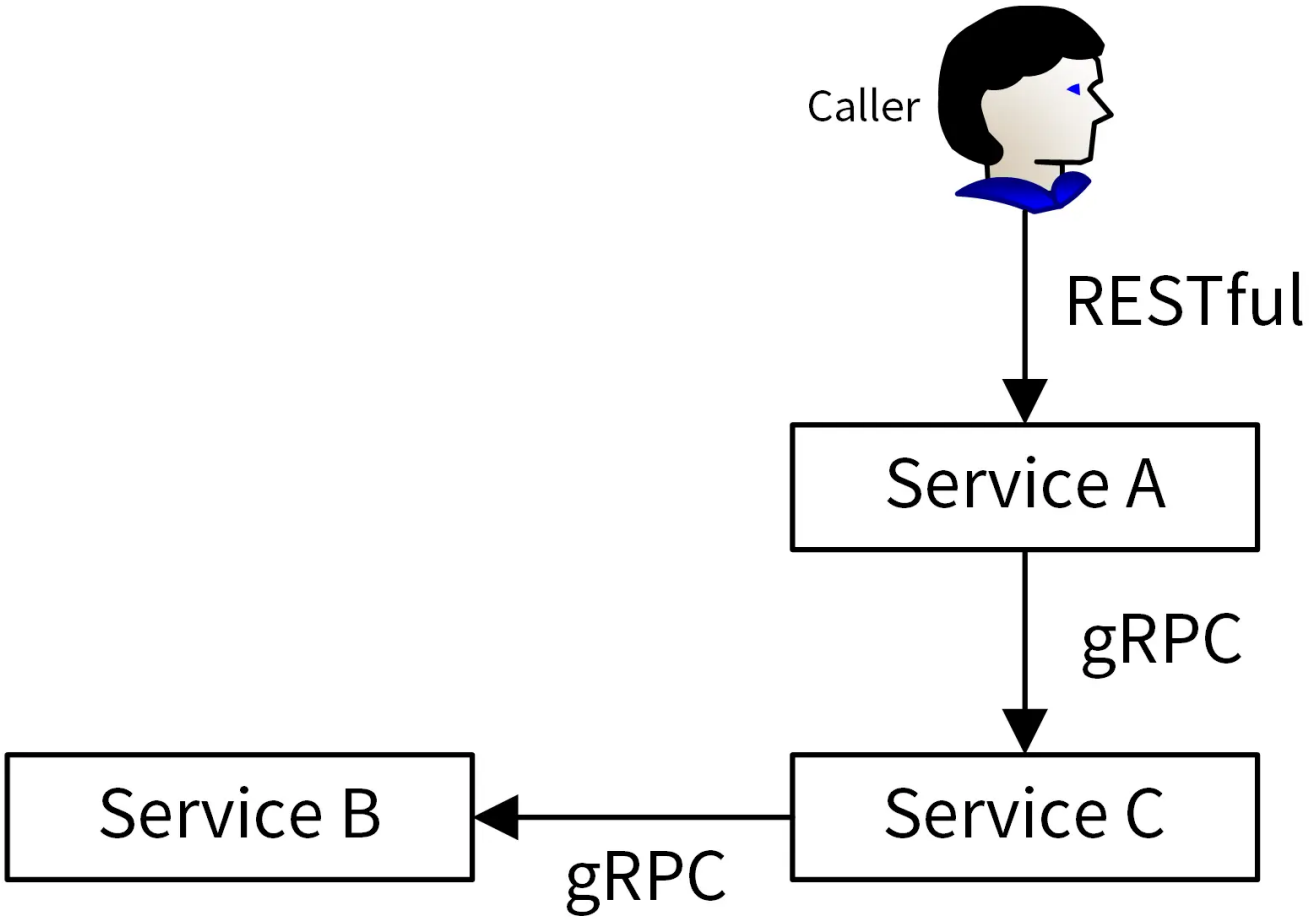
总之，在 GetUser 方法中，我们可以通过判断 r.Username 是否为 nil，来判断客户端是否传入了 Username 参数。

RESTful VS gRPC

到这里，今天我们已经介绍完了 gRPC API。回想一下我们昨天学习的 RESTful API，你可能想问：这两种 API 风格分别有什么优缺点，适用于什么场景呢？我把这个问题的答案放在了下面这张表中，你可以对照着它，根据自己的需求在实际应用时进行选择。

对比项	RESTful	gRPC
优点	• 相对来说，REST更规范，更标准，更通用，无论哪种语言都支持HTTP协议，可以对接外部很多系统，只要满足HTTP调用即可； • RESTful接口通常采用JSON作为数据通信格式，JSON格式可读性强，易于理解，开发调试都很方便； • 客户端和服务端松耦合，让变更更加方便	• 屏蔽网络细节，调用远程接口像调用本地方法一样，调用简单、方便； • gRPC往往会采用Protocol Buffers作为数据传输格式，这种格式数据传输效率高； • gRPC基于HTTP/2协议标准，性能更高
缺点	• 扩展性差：随着需求的变化，单个RESTful接口可能会变得越来越臃肿； • 性能相对于gRPC偏低	• Protobuf数据格式可读性差； • gRPC不支持浏览器调用，因此不能用作外部服务； • gRPC调试不方便，没有像RESTful中的postman和curl之类的调试工具
使用场景	• 接口对外，需要接口规范易懂； • 对系统性能要求不高； • 提供的API天生围绕资源、对象、管理展开	• 消息密集型、对系统性能和延时要求比较高； • 偏向内部的API； • 提供的API很难进行资源、对象抽象

当然，更多的时候，RESTful API 和 gRPC API 是一种合作的关系，对内业务使用 gRPC API，对外业务使用 RESTful API，如下图所示：



总结

在 Go 项目开发中，我们可以选择使用 RESTful API 风格和 RPC API 风格，这两种服务都用得很多。其中，RESTful API 风格因为规范、易理解、易用，所以**适合用在需要对外提供 API 接口的场景中**。而 RPC API 因为性能比较高、调用方便，**更适合用在内部业务中**。

RESTful API 使用的是 HTTP 协议，而 RPC API 使用的是 RPC 协议。目前，有很多 RPC 协议可供你选择，而我推荐你使用 gRPC，因为它很轻量，同时性能很高、很稳定，是一个优秀的 RPC 框架。所以目前业界用的最多的还是 gRPC 协议，腾讯、阿里等大厂内部很多核心的线上服务用的就是 gRPC。

除了使用 gRPC 协议，在进行 Go 项目开发前，你也可以了解业界一些其他的优秀 Go RPC 框架，比如腾讯的 tars-go、阿里的 dubbo-go、Facebook 的 thrift、rpcx 等，你可以在项目开发之前一并调研，根据实际情况进行选择。

课后练习

1. 使用 gRPC 包，快速实现一个 RPC API 服务，并实现 PrintHello 接口，该接口会返回“Hello World”字符串。
2. 请你思考这个场景：你有一个 gRPC 服务，但是却希望该服务同时也能提供 RESTful API 接口，这该如何实现？

期待在留言区看到你的思考和答案，也欢迎和我一起探讨关于 RPC API 相关的问题，我们下一讲见！

分享给需要的人，Ta订阅后你可得 24 元现金奖励

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | API 风格（上）：如何设计RESTful API？

下一篇 14 | 项目管理：如何编写高质量的Makefile？

精选留言 (12)

写留言

pedro
2021-06-24

假定希望用RPC作为内部API的通讯，同时也想对外提供RESTful API，又不想写两套，可以使用gRPC Gateway 插件，在生成RPC的同时也生成RESTful web server。

展开

作者回复: 正解

7

柠柠
2021-06-25

RPC 与 RESTful 共通逻辑抽象出来 Service 层，RPC server 和 RESTful server 初始化or启动时时都需要指定 service，真正提供服务的是 Service 层

展开

作者回复: 对的！

1

A

2021-06-25

https://www.bookstack.cn/read/API-design-guide/API-design-guide-04-%E6%A0%87%E5%87%86%E6%96%B9%E6%B3%95.md 我又来做贡献了

展开

作者回复: 感谢分享，文章非常棒！已收藏！

1

漂泊的小飘

2021-07-21

老师，有两个问题请教下：
1，能否讲下四种模式的应用场景？
2，一般生成的文件需要放进git里面吗

展开

张名哲

2021-07-13

老师，文章中有四种模式，平时用的最多的是哪一种模式？

展开

作者回复: 简单模式用的比较多

learner2021

2021-06-24

哪里设置错误了？
[going@dev server]$ pwd
/home/going/workspace/golang/src/github.com/marmotedu/gopractise-demo/apistyle/greeter/server
[going@dev server]$ go run main.go...

展开

作者回复: 看看github.com/marmotedu/gopractise-demo目录下，有没有go.mod。

如果有建议执行下：go mod tidy

1

kkgo
2021-06-24

老师有对比过grpc和rpcx之间的性能，稳定性方面不? 看官方说明rpcx性能是grpc的2倍

作者回复: 没对比过，老哥可以对比下性能，分享出来

helloworld
2021-06-24

优秀👍

展开

Star°时光℡
2021-06-24

protoc -I. --go_out=plugins=grpc:$GOPATH/src helloworld.proto
执行命令后无任何输出，也没有产生helloworld.pb.go文件

展开

作者回复: 输入信息有点少，不清楚为啥你的命令没有生成文件，可能跟环境有关。

也可以试试这篇教程：https://developers.google.com/protocol-buffers/docs/gotutorial。

Star°时光℡
2021-06-24

protoc -I. --go_out=plugins=grpc:$GOPATH/src helloworld.proto
```

[root@dev helloword]# ls
```

helloworld.proto
[root@dev helloword]# cat helloworld.proto ...
展开

夏夜星语
2021-06-24

之前自己还用jsonrpc 写rpc, 现在用这个感觉完全接口更规范，更简单

不明真相的群众
2021-06-24

你有一个 gRPC 服务，但是却希望该服务同时也能提供 RESTful API 接口，这该如何实现？

在封装一层？

作者回复: 用这个：https://github.com/grpc-ecosystem/grpc-gateway